Muon Reconstruction Review

Jim Kowalkowski
Marc Paterno

1 Introduction

This review considered the various packages related to muon reconstruction, excluding those that are provided the **trf++** tracking package. The CVS packages were

- **muo_data**
- **muo_dataprocessor**
- **muo_hitreco**
- **muo_segmentreco**
- **muo_trackreco**

We considered both design issues and implementation issues, and address these in separate parts of this document. Our talk from the July DØ workshop in Seattle was used as a guideline for our discussion of use of the EDM and of the framework.

This document looks into the relationship of algorithm objects to data objects. Due to time considerations, only the higher level data objects that appear in the event were considered. The intermediate segment finding data objects such as LocalWireHit were only looked at briefly. Other areas this document covers are C++ implementation issues such as memory management and code efficiency.

There are several areas we do not address in this document: the muon geometry classes, and the packages **muo_utils**, **muon_index** and **spacegeom**.. We have omitted any analysis of these items for lack of time; there are, however, passing comments concerning obvious deficiencies in these packages.

2 Overview

We were very happy to receive clear class diagrams and a clear high-level overview of the system. This made the task of reviewing the software considerably easier than it would otherwise have been, and probably saved two or three days' effort.

It is our current understanding that the algorithm code itself must be able to run in level 3 and in the offline environments. This means that the algorithm cannot have direct visibility to the EDM classes (Chunk, Event), or the framework classes (Package), or the level 3 "tool" classes. The organizational concepts present in the muon packages appear to be correct at the highest level. That is to say, the breakdown of offline reconstructor package orchestrating the reconstruction by plucking items from the event, creating and using algorithm objects it appropriate. The level 3 tools can do the same orchestration without the use of the event or chunks if one is careful. Our intention was to produce a figure that illustrates our current understanding of the system at a high level and the pieces that this report focuses on. Due to time constraints this is not possible. The talk from Onne Peters can be used for this purpose.

The "MuoDataProcessor" concept discussed at the review meeting seems to only appear in the hit finding parts of the system. This concept is not employed in the segment and track finding parts. We were under the impression after the review meeting that this was a fundamental principle that all the code was based on. We also found that the current hit finding code does not do anything with the scintillator hits.

2.1 Definitions

Adaptor Class: A class that extends the functionality of another simpler class, not by inheritance, but by containment. The containment is not by value but by pointer or reference. An adaptor class in this context takes the object that it is extending as an argument to the constructor. The methods of the adaptor class call down to the contained class and typically provide higher level services.

3 Major Concerns

We have several significant concerns regarding this software. These concerns have been divided into several different groups:

1. Coding concerns: this includes matters of program correctness, safety and efficiency in the use of C++.
2. Design concerns: this concerns the larger-scale issues of design, especially of the appropriate use of object-oriented and generic programming techniques, and appropriate use of DØ infrastructure code.
3. Management concerns: this includes matters of the software development and management process, and adherence to the DØ software development guidelines.

3.1 Coding

3.1.1 Memory Management

Memory management is a disaster. When objects are created with *new*, it is critical that the ownership of the object is clear at all times so that the memory can be cleaned up. Knowing the owner ensures that all objects are destroyed and destroyed only once. Knowing this also will make sure that items are shared using pointers and references only when sharing is actually intended. There appears to be a fairly consistent memory management policy throughout the code. We understand this policy to be:

> Containers and objects are always created with new. Objects created by functions are returned by pointer and it is the callers' responsibility to clean up the memory. Containers mostly hold pointers to objects. Copying containers only copies the pointers.

We believe that this is not the proper memory management scheme for the problem. We further believe that this scheme is the reason for the existence of the D0OM flavored objects that live in the event. This scheme produced many inconsistencies and ambiguities in construction of the containers (copy vs. make empty one). In most cases, the policy is followed for the creation of objects, and the memory is never recovered by the calling functions. The number of places this occurs is too numerous to include in this document. This is probably the most important item that needs to be resolved and fixed. This will cause the reconstruction executable to crash. This policy is likely to cause the code to perform very poorly, partly because of the heavy copying and heavy use of the memory allocator.

Using this policy causes other subtle problem to appear, such as this piece of code in the track finding algorithm:

```
MuoSegmentMatchList xslist;
…
// Sort list by matching quality
xslist.sort();
```

The problem here is that this is a list of pointers. Sorting it does not sort by matching quality, it sorts by ascending pointer value, which is typically useless and in error. The second problem here is that sorting a list is very inefficient.

3.1.2 Error Reporting

There are basically two types of errors: global errors that affect the behavior of the program overall, and local errors that only affect a single reconstructor or algorithm. A potential global error, for example, could be that a data integrity check-sum has been used to detect an error in the current event data. The code that detected the error will need to decide if the event should be considered by packages downstream of itself, or if it should be throw away. This type of error impacts all the reconstuctors that are active. A local error, for example, can be generated by a portion of the algorithm code that report if too many objects of a particular type have been produced from this event. This error may not be significant to any other reconstructors and not affect the program behavior at all. Both of these are not handled in the muon code.

The first type of error is a problem for all reconstruction packages, because a policy has not been established by the infrastructure group. Currently attempts are made to return error codes and print simple messages to the console. Both of these are bad. Printing messages to *cout* just causes confusion, especially if the message does not contain context. In most cases the return codes are not properly propagated up the call change where decision about continuing can be made.

3.1.3 Miscellaneous

When a reference to a chunk is retrieved from the event into a Thandle<>, it should never be removed from the Thandle<>. Under no circumstances should it be removed. Many of the reconstructors pull the chunk pointers from the Thandle<>. Some of the extractions actually cause the *const* nature of the object to be lost. The Thandle is there to protect you and the information in the event, extracting its contents defeats this purpose.

In several places we found constants that were hard-coded into the algorithm implementation files. Many of these constants should be retrieved via RCP or from a geometry/hardware configuration database. One should avoid coding any constants into the application that may need to be changed in the future or that may vary with time. These constants are typically impossible to track down when changes need to be made.

3.2 Design

3.2.1 Template Strategy and Algorithm Objects

The use of a class template for MuoDataProcessor<> provides no gain at the great expense of complexity. All of the MuoDataProcessors, as used in the algorithms, are really just functions. The normal overloaded function mechanism of C++ would suffice. The functions that comprise an algorithm really need to be composed into an algorithm object; one that can be configurable through RCP or another means.

Templates are good for expressing commonality of structure between many classes (algorithms in this case). The only commonality expressed by the MuoDataProcessor template is that the single static method has the same “processData” and that it takes two arguments and returns one thing. This is much too trivial a requirement to consider templates as a viable solution.

The MuoSegmentReco package appears to use a more appropriate strategy. Here there is an abstract base class for the algorithm and a concrete implementation. The reconstructor creates the algorithm, which can perhaps be chosen by looking at an RCP value, then invokes it to carry out the work. We are hoping this model can be expanded upon in favor of the MuoDataProcessor model.

It is unclear exactly what the MuoTrackReco package will be doing in this respect. We found differences in strategies between t00.67.00 and t00.68.00 that could not be explained. The strategy from the ‘68 version appeared to be moving towards the MuoSegmentReco model. The strategy of the ’67 version appears to be a Fortran-like model. We did not spend very much time looking into this area.

3.2.2 Problem Decomposition

In looking through the algorithms, we had a difficult time telling what exactly the algorithm was and what it was actually doing. One of our concerns is that we could not tell what manipulations are completely defined by the detector design (fixed angle calculation, fixed relationships of components) and which were actually algorithmic, or higher level items. The specific example we use in this document is the PDT hit finding function. Here the loop through timing values in the channel looks at all pairs of wires in the channel. We could not discern, for example, what combinations of wires were physically possible and which were not, or if the angle and point calculations were properties of the channel or strictly determined by the algorithm (if the calculation could various between algorithms that make wire hits from PDTChannels). What we would have like to have seen would be a clear separation of things that go into making decisions about what is a wire hit from things that do simple calculations. The calculation here being defined as one that has to do entirely with the design of the detector and would not change if the wire hit finding algorithm changed. We do not mean to pick on this particular piece of code; it was a small enough piece for us to go over and discover this problem. Most of the code suffers from the same problem.

The source of this problem is that the algorithm has not been decomposed properly into manageable units. Many of the tasks are performed directly by the algorithm code, as opposed to utilities and tools. The tasks perform operations on dumb data structures rather then real objects with behavior.

3.2.3 Hit Storage

The hit storage container MuoHitCollection appears to be not very useful as it stands. Having the two multimaps appear together all the time does not seem to be a good idea, judging from how the algorithms are written. We are under the impression that in some sections of the detector, the hits referred to in tracks can be either wire hits or scintillator hits. The current MuoHitCollection does not facilitate this effort and does not provide adequate services to perform this function. The current design will most likely lead to more of the problems described in the previous section. Another problem that will likely occur will be when segments currently refer to the hits they contain by MuoIndex only. The only way to determine if it is a ScintHit or a WireHit would be to check the bits in the MuoIndex, and then go to the correct multimap in the MuoHitCollection. This is not a clean way for users of the segments and tracks to navigate to the hit objects.

3.2.4 D0OM Object Versions

Throughout the code, we could not see a need to have persistent derivations of basic data objects. Doing this creates more classes then are needed and forces copies of many objects to occur. We believe that this strategy exists because of the memory management policies and because algorithms return newly created objects.

3.2.5 EDM and Framework Usage

The muon reconstruction code makes poor use of the Event Model. There are no interesting functions in the event data classes (chunks) and no selectors to find them. The talk that we gave at the Seattle D0 workshop discusses good chunk design and should be used as a starting point. The reconstructors (framework packages) that produce chunks do not fill in important details into the chunk such as parentage. The reconstructors that find chunks in the event do not use keys in any interesting way and do not use selectors. Locating objects in the event as the reconstructors do will essentially return any chunk of that type. If more than one chunk exists of that type in the event, then there is no guarantee you will get the correct chunk.

Segments refer to hits using a vector of MuoIndex. It will be difficult to tell if the hit is a scint or wire hit. More tools will be needed for extracting scint or wire hits if this is the strategy that will remain.

3.2.6 Use of Common D0 Tools and Utilities

In several places the standard D0 tools are not being used or the standard tools are being replaced by custom ones. Section 4.1 on recommendations contains a detailed discussion about this.

3.3 Testing

The component tests throughout the muon code are mostly dummies. This is really not acceptable. Section 8 has a more detailed explanation of testing.

4 Design Recommendations

Several of the design issues that we found in this set of packages are present in other D0 packages.

4.1 Common D0 libraries and Tools

There is a large amount of low-level utility code available to developers at D0. In many cases we find that developers re-implement these utilities or work around missing features by producing large amounts of code directly in the algorithm. In other cases developers simply do not know that utilities and tools exist. We see this problem in the muon code. It is not possible to cover every case, but we can discuss a few that we have found. We use these three examples because each illustrates a different type of problem:

(1) perhaps dislike for a particular set of classes, so re-implementation occurs,

(2) missing features, so ugly code appears surrounding its use, and

(3) inadequacies in a class cause duplication of effort.

4.1.1 SpacePoint / CartesionPoint

In many sections of the code, points in space are represented by an array of three doubles. A standard package called **spacegeom** exists that has several classes designed to represent points in space. Working with a standard object is the correct thing to do, so all users can share an understanding of how the object works and feel comfortable looking through code that is not part of their subsystem. The muon_geometry actually uses the SpacePoint and CartesionPoint objects, but then chooses to pick the information out and present it in a different representation.

If the classes in the **spacegeom** package are not adequate because of memory usage, performance, or interface, then the owners of that package should be approached so that changes can be made to the classes. The use of a non-object such as an array of doubles is particularly bad; you cannot even do simple operations such as copy the point unless you hand-code a loop. Copying the point (array of doubles) in this case is a huge distraction from what the algorithm is actually doing and a potential spot for a bug.

4.1.2 UnpDataChunk

The extraction of channels from modules in the UnpDataChunk in the muon code is quite hideous and unpleasant (like kissing alpaca lips). It is hideous in terms of understandability, resilience to change, processing time, and memory usage. The bit manipulations make it difficult to change the underlying classes – because you have intimate knowledge of internal details in the high-level application code. The PDTChannels and MDTChannels, for example, are hefty and require quite a bit of work and memory to copy.

Talking to the Author of UnpDataChunk about the PDTChannel and MDTChannel use case could greatly simplify your life and this code. Adding two small utilities that allow for extraction of channels and modules of a particular type from the UnpDataChunk - by pointer, not by value – could reduce the code in the algorithm down to two or three lines that are clear and easy to understand.

4.1.3 MuoIndex

After the review result meeting on 12/16/1999, we had the impression that MuoIndex was not going to be used in the level 3 algorithm because of its size. This means that the algorithm will be completely different code in level 3 that does the exact same thing because the offline requires the use of MuoIndex. This is a shame.

If size is the major concern here, then a simple conversion with Mike Fortner could show that this object can be reduced in size by perhaps nine times if need be. If the MuoIndex is not adequate for level 3, then the owner of MuoIndex should be approached to see if it can be made acceptable for level 3.

4.2 Classes with Useful Features

There are many classes in the muon packages that are essentially featureless data structures. In many cases, these classes can be made to do work that the algorithms and user of them need. A simple example is the ListOfPoints class in the segment finding algorithm. Here the ListOfPoints is just an STL vector<LocalPoint*>. When looking through code that uses this class, it quickly becomes clear that the user code must implement functionality that should really be part of the class itself. An example is the markHits() method of the MuoSegementAlgCombi class. Here marking all the hits used in the ListOfPoints should be a method of the ListOfPoints class. Inheritance can be used to extend the functionality of the vector and list classes to add things like markHits(). Moving markHits() to the ListOfPoints class puts this function where it belongs and allows this action to occur correctly with a single method call. Many of the collection classes (typedefs) suffer from this problem.

4.3 Example MuoHitProcessor Changes

We were going to present a couple of snapshots of the MuoHitProcessor code, each showing specific changes that can be made. Unfortunately that would take more time than we have. Instead, the final version parts are shown below, along with explanations of what they do. The intention is not to show exactly how this code should be modified; rather, it is to show what sort of modifications are needed to many parts of the code, to enhance the maintainability of the code and to easily track down problems. This describes how to do away with the hit processing class templates, and to replace them with meaningful configurable algorithm objects. Much of the MuoHitProcessor is used here and as the example in this document. A few examples are given from the segment finding algorithm to illustrate to sort of things that should be done there. The hope here is that these MuoHitProcessor changes can be used as a model for the segment and track finding algorithms. This piece of algorithm code was chosen because it does should not involve chunks or any EDM related classes. The EDM related pieces of the system are left to another section of this document.

4.3.1 Example MuoHitCollection Changes

In the current design the MuoHitCollection is really just a pair of multimaps put together with a couple utility methods to aid in the insertion of hits. Other then this, the maps are really used separately by the code. Judging from the fact that hits are referred to in segments by MuoIndex only, we made the assumption that the most interesting high-level use of hits will be “give be all the wire hits associated with this MuoIndex” or “give be all the scint hits associated with this MuoIndex” or “give me all the MuoHits associated with this MuoIndex. If this is true, then giving the user a list or vector of hits seems appropriate and natural instead of a beginning and ending iterator of a mulitmap. Our example here assumes that this is how one wants to operate with hits. Similar techniques can be used even if our thinking does not exactly match reality.

The first thing we will do is separate the two types of hits into separate objects and provide functions to aid in the insertion hits. Notice that the base object here that hits are stored in is a map of MuoIndex -> vector of hits.

```
template <class HIT, class CONT = std::vector<HIT> >
class HitContainer : public std::map<MuoIndex,CONT> {
public:
    typedef HIT hit_type;

    HitContainer() { }

    bool insertHit(const HIT& hit) {
      pair<iterator,bool> I(find(hit.index()),true);
      if(I.first==end())
```

```
        I=insert(value_type(hit.index(),data_type()));
      if(I.second==true)
        I.first.push_back(hit);
      return I.second;
    }
};

typedef HitContainer<MuoWireHit> WireHitMap;
typedef HitContainer<MuoScintHit> ScintHitMap;

// this could be a method of the HitContainer class
template <class CONT, class RET>
void extractAsHits(const CONT& cont, RET& fillme) {
    for(CONT::const_iterator it=cont.begin();it!=cont.end();++it)
        fillme.push_back(&(*it).second);
}

void use_example_func(const WireHitMap& wh, const ScintHitMap& sh) {
    vector<MuoHits*> shared_common;
    extractAsHits(wh,shared_common);
    extractAsHits(sh,shared_common);
    sort(shared_common.begin(),shared_common.end());
    ... use all the hits in there base format ...
}
```

It is likely that organization such as this and this use of templates is overkill for a small part such as collecting hits.

4.3.2 Example Channel Adapter Utility Classes

As we looked through the PDT and MDT hit finding code, we immediately noticed that extended channel classes could help out quite a bit. Two new classes are introduced here to assist in channel related operations. Both are derived from a common base class.

```
class HitChannel
{
public:
    HitChannel(const Channel& c) {...set private info...}

    const MuoIndex& index() const { return _index; }
    const MuoSectionIndex& sectionIndex() const { return _section; }
    const CartesionPoint& position() const { return _pos; }
    const CartesionPoint& orientation() const { return _ori; }

protected:
    HitChannel(const Channel& c, const MuoIndex& m) {...set private info...}

    const Channel* _chan;
    MuoIndex _index;
    MuoSectionIndex _section;
    CartesionPoint _pos;
    CartesionPoint _ori;

    static MuoIndexTrans _index_trans;
};

class TimePairs
{
public:
    TimePairs():_t1(0.),_t2(0.)._has_info(false) { }
    TimePairs(double t1, double t2, double wire_len,double angle):
        _t1(t1),_t2(t2)
        { _has_info=T1T2ToTDTAPDT(_t1, _t2, wire_len,_td,_ta) &&
            TDTAToDrDistAxDistPDT(_td,fabs(_ta),angle,_xd,_xa); }

    // just guested at the appropriate names here
    double driftTime() const { return _td; }
    double axialTime() const { return _ta; }
    double driftDistance() const { return _xd; }
    double axialDistance() const { return _xa; }

    bool hasInfo() const { return _has_info; }
private:
    double _t1,_t2;
    double _xd, _xa, _td, _ta;
    bool _has_info;
};

class HitPDTChannel : public HitChannel
{
public:
    HitPDTChannel(const PDTChannel& c):HitChannel(c) {...}

    const MuoGeomPDT* getPDT() const;
    double wireLength() const;
    double axDistToTime() const;
    double angle() const;
    HitPDTChannel next();

    // this is only a sample - can be done much better (should not be inline)
    void validTimePairs( vector<TimePairs>& fillme ) const {
        fillme.clear();
        double angle = chan.angle();
        double len = chan.wireLength();
        for(int i1=0;i1<chan.numEvenTimes();++i1) {
            for(int i2=0;i2<chan.numOffTimes();++i2) {
                TimePair t(chan.timeEven(i1),chan.timeOdd(i2),len,angle);
                if(t.hasInfo()==true) fillme.push_back(t);
            }
        }
    }

private:
    const MuoGeomPDT* _pdt;
};

class HitMDTChannel : public HitChannel
{
public:
    HitMDTChannel(const MDTChannel& c):
        _hit_wire(..calc...),HitChannel(c){...}

    static double binSize() { return 18.8; } // is this really fixed?

    const MuoGeomMDT* getMDT() const;
    const MuoIndex& wireIndex() const;
    const CartesionPoint& wirePosition() const;
    const CartesionPoint& wireOrientation() const;
    double wireLength() const;
    double driftDistance() const;
    double tof() const;
    double wireLength0() const;
    double wireDl() const;
    double distanceToOrigin() const;
    double driftTime() const;

private:
    MuoIndex _wire_index;
    int _hit_wire;
    const MuoGeomMDT* _mdt;
    CartesionPoint _wposition;
    CartesionPoint _worientation;
};
```

4.3.3 Example HitBuilder Algorithm Object

As mentioned in the concerns and problems section, an algorithm object is needed to replace the "MuoDataProcessor" template and functions. The functions that find the UnpDataChunk and use the Event are removed; the code that does this should live directly in the reconstructor. Here is an example of how the PDT and MDT hit finding code can be represented by an algorithm object. It is important to note that we assumed that classes such as CartesionPoint support multiplication by a scalar or another point, and also support addition and subtraction – after all, this is one of the benefits of using C++.

```
class HitBuilder {
public:
  typedef const vector<const PDTChannel*> PDTChannels;
  typedef const vector<const MDTChannel*> MDTChannels;

  HitBuilder(RCP r);
  virtual ~HitBuilder() { }
  virtual void buildWireHits(PDTChannels& chans, WireHitMap& hits) = 0;
  virtual void buildWireHits(MDTChannels& chans, WireHitMap& hits) = 0;

  void buildWireHits(PDTChannels& c1, MDTChannels& c2, WireHitMap& hits)
    { buildWireHits(c1,hits); buildWireHits(c2,hits); }

  // virtual void buildScintHits(...) = 0; // future use

private:
  static double speedOfLight() { return 29.98; } // cm/ns
  // just assume that all these come from RCP
  double v_drift;
  double v_axial;
  double drift_error;
  double max_time_dif;
  double x_d_max;
  double delay;
};

class SimpleHitBuilder : public HitBuilder { ... };

// The implementation of the wire hit building using PDTChannels could be as follows:
SimpleHitBuilder::buildWireHits(PDTChannel& chans, WireHitMap& hits) {
    vector<TimePair> pairs;
    PDTChannel::const_iterator iter = chans.begin();
    for(;iter!=chans.end();++iter) {
        HitPDTChannel chan(*iter);

        if( chan.getPDT()==0 )
        {
            // log an error using the error logger
            // assume that channel data is bad and that we cannot continue
            // throw specific exception
        }

        chan.validTimePairs(pairs);
        double wire_len_cent;
        double drift_error = 0.1*xd;
        vector<TimePair>::iterator titer = pairs.begin();

        for(;titer!=pairs.end();++titer)
        {
            drift_error=0.1 * (*titer).axialDistance();

            if((*titer).axialTime() <=0 )
            {
                HitPDTChannel nchan(chan.next());
                wire_len_cent = .5 * nchan.wireLength() - (*titer).axialDistance();

                hits.insertHit(MuoWireHit(
                    Position(
                        nchan.position() - (wire_len_cent * nchan.orientation()),
                        _axial_time_error * nchan.orientation(),
                    ),
                    nchan.index(),
                    (*titer).driftTime(),
                    (*titer).driftDistance(),
                    driftError
                ));
            }
            else
            {
                wire_len_cent = .5 * chan.wireLength() - (*titer).axialDistance();

                hits.insertHit(MuoWireHit(
                    Position(
                        chan.position() - (wire_len_cent * chan.orientation()),
                        _axial_time_error * chan.orientation(),
                    ),
                    chan.index(),
                    (*titer).driftTime(),
                    (*titer).driftDistance(),
                    driftError
                ));
            }
        }
    }
}

// The implemenation of the wire hit building using MDTChannels could be as follows:
SimpleHitBuilder::buildWireHits(MDTChannel& chans, WireHitMap& hits) {
    MDTChannel::const_iterator iter = chans.begin();
    for(;i!=chans.end();++i) {
        HitMDTChannel chan(*iter);

        if( chan.getMDT()==0 )
        {
            // log an error using the error logger
            // assume that channel data is bad and that we cannot continue
            // throw specific exception
        }

        if(chan.noTime()==true)
        {
            // Is this it for the message ?????  Is this an error?
            // "No time in MDT tube"
        }
        else
        {
            // Here we leave the pos_error and drift_err calculation in the algorithm
            // instead of letting the channel class do it.  This may not be
            // necessary.
             double driftdist_err = 0.1 * chan.driftDistance();

            hits.insertHit(MuoWireHit(
                Position(
                    chan.wirePosition(),
                    (.5 * chan.wireLength()) * chan.wireOrientation()
                ),
                chan.index(),
                chan.driftTime(),
                chan.driftDistance(),
                driftdist_err
            ));
        }
    }
}
```

4.4 EDM Use

The design tutorial present by us at the Seattle D0 workshop should be used as a simple guide. Here are some of the key issues:

- *D0OM allows STL collections of objects (not pointers) to persist automatically without the need to inherit from D0_Object. With the proper reorganization and memory management plan, all of the persistent collections should be able to fit into this model. This has already been discussed briefly and will be elaborated on in following sections.*
- *There are other options for referring to collections of hits at the segment level and collections of segments at the track level. The LinkVectorIndex<> and LinkVectorPtr<> classes may simplify the use of the muon chunks.*
- *If transient versions of information that lives in a chunk are needed, such as collections of pointers to hits, then use of the D0OM utilities for creating transient data automatically may be needed such as activate() and deactivate().*
- *Redesign of the MuoHitCollection to separate the two maps. The Chunk provides utilities to gather pointers from both maps into one vector or list of pointers (to the base hit class). Part of this has already been discussed earlier in this document.*

A muon hit chunk to support many of things discussed in this document could look as follows. A chunk organized like MuoHitChunk below allows the use of the edm links. Unfortunately, because of level3 requirements, the segment and track chunks may not be able to take advantage of this without a bit more work than can be presented in this document. The second chunk in this code segment shows how one might refer to hits in the MuoHitChunk using the edm link classes. The important thing here is the first MuoHitChunk, the HitUser* examples can be looked at as a secondary issue.

```
class MuoHitChunk : public edm::AbsChunk
{
  CHUNK_SETUP( MuoHitChunk );
public:
  // full service chunk for use with edm links
  typedef std::vector<MuoHit*> CommonHits;

  MuoHitChunk();
  ~MuoHitChunk();

  list<edm::ChunkID> parents() const;
  list<edm::RCPID> rcps() const;
  list<edm::EnvID> environment() const;
  void printChunk( ostream& out ) const;

  // access separate pieces
  const WireHitMap&  wireHitContents() const { return _wire_hits; }
  WireHitMap& wireHitContents() { return _wire_hits; }
  const ScintHitMap& scintHitContents() const { return _scint_hits; }
  ScintHitMap& scintHitContents() { return _scint_hits; }

  // should include these standard access methods
  WireHitMap::data_type& wireHitAt(WireHitMap::key_type index);
  ScintHitMap::data_type& scintHitAt(ScintHitMap::key::type index);
  CommonHits::value_type& at(CommonHits::size_type index);

  // allow for more than one type of object lookup...
  struct WireHitLookup {
    const WireHitMap::data_type* operator()(const MuoHitChunk* c,
        WireHitMap::key_type i)
          { return &(c->wireHitAt(i)); }
  };
  struct ScintHitLookup {
    const ScintHitMap::data_type* operator()(const MuoHitChunk* c,
        ScintHitMap::key_type i)
          { return &(c->scintHitAt(i)); }
  };
  // treat all hits the same
  CommonHits& contents()
    { if(_allready==false) buildHits(); return _all_hits; }
  const CommonHits& contents() const
    { if(_allready==false buildHits(); return _all_hits; }

private:
  void buildHits() const {... put all hits in _all_hits and set flag... }
  WireHitMap _wire_hits;
  ScintHitMap _scint_hits;
  mutable CommonHits _all_hits;
  mutable _allready;
};

// example of how a user chunk MIGHT look, using the edm links
class HitUserSegment
{
public:
  edm::LinkIndexVector<MuoHit> HitLinks;

  HitLinks& hitContents() { return _hits; }
  const HitLinks& hitContents() const { return _hits; }
  HitList::IndexList& contents() { return _hits.contents(); }
  const HitList::IndexList& contents() const { return _hits.contents(); }

  void setHitChunk(edm::AbsChunk* c) { _hits.setLinkValues(c); }
  void completeLinks(const edm::AbsChunk* c) const
    { edm::finishUpLink(hits,c); }
  ...
private:
  HitLinks _hits;
  CartesionPoint  _position;
  CartesionPoint  _direction;
  double          _quality;
  double          _angleDr;
};
typedef std::vector<HItUserSegment> HitUserSegments;

class HitUserChunk : public edm::AbsChunk
{
  CHUNK_SETUP(HitUserChunk);
public:
  HitUserChunk();
  ~HitUserChunk();

  ... standard chunk stuff ...

  HitUserChunk::value_type& at(HitUserChunk::size_type index)
    { doLinks(); return _segments[index];}
  const HitUserChunk::value_type& at(HitUserChunk::size_type index) const
    { doLinks(); return _segments[index]; }
  HitUserSegments& contents()
    { doLinks(); return _segments; }
  const HitUserSegments& contents() const
    { doLinks(); return _segments; }

private:
  void doLinks() const
  {
      if(linksDone==true) return;
      for(HitUserSegments::const_iterator i=_segments.begin();
           i!=_segments.end();++i)
          (*i).completeLinks(this);
      linksDone=true;
  }

  HitUserSegments _segments;
#ifndef __CINT__
  mutable bool linksDone;
#endif
};

// example use of this HitUserChunk object
void func(THandle<HitUserChunk> uchunk)
{
  const HitUserSegments& segs = uchunk->contents();
  HitUserSegment::const_iterator i = segs.begin()
  for(;i!=segs.end();++i)
  {
      LinkPtrVector<MuoHitChunk,MuoHit> uv((*titer).hitContents());
      LinkPtrVector<MuoHitChunk,MuoHit>::const_iterator hiter;

      for(hiter=uv.begin(); hiter!=uv.end(); ++hiter)
        cout << (*hiter)->position() << endl;
  }
}
```

As mentioned in the section on concerns, the selection of chunks from the event is not adequate. Most of the code looks like the following:

```
  // Get MuoHitChunk.
       const TKey<MuoHitChunk> muonhitKey;
       THandle<MuoHitChunk> ptrmuonChunk=muonhitKey.find(event);

       if (ptrmuonChunk.isValid()){
          …
       }
```

This will find ANY chunk of that type in the event, not necessarily the one that is desired. The chunks should at least have some distinguishing feature that one can use, such as algorithm name and version, or RCPID. If you are unsure, all the chunks of a given type should be retrieved, so a fair decision can be made as to which one to use. When a chunk such as MuoSegmentChunk is created, it must record the ChunkID of the MuoHitChunk that was used to construct it. This is the only way to ensure that when users want to go back to the hits that are contained in the segment, they can be sure to get the correct Muo-HitChunk.

4.5 Framework Use

The event processing framework has evolved some over the past year. None of the static methods in the reco classes are required and should be removed. There are two important points that need to be made in this section. One is that reconstructor should do some of the work, such as locating chunks in the event and driving the algorithm. Activities that are related to working directly with the framework, the EDM, or D0OM can be done directly in the framework package. The reconstructors (packages) can be tested outside the framework main routine. Another item of interest here is what the muon hit processing reconstructor could look like given the proposed changes.

4.5.1 Testing

The framework has a feature that allows one to construct a framework package outside the framework main program. This feature is useful for testing reconstructors (framework packages) that you write. This facility creates an instance of your package and passes it back to you. You can call the processEvent method or any other method yourself to do testing. Here is an example of its use:

```
#include “MyPackage.hpp”
#include “framework/Testing.hpp”
int main(int argc, char* argv[]) {
   string rcp_info = argv[1];
   MyPackage* mine;
   Fwk::makePackage(mine,rcp_info);
   … open a test input file or prepare a test event somehow …
   Event* e = getEvent(…);
   mine->processEvent(e);
   … dump results …
   delete mine;
}
```

4.5.2 Example Reconstructor

Here we present a major part of what the muon hit processing framework package could look like. We would have liked to do a similar job with the segment and track reco packages, but it would take too much time.

```
MuoHitReco::MuoHitReco(Context* c):Package(c)
{
    string algo = packageRCP().getString("Algorithm");

    if(algo == "Standard")
        _hit_builder = new SimpleHitBuilder(packageRCP());
    else
    {
        error_log(ELabort,"init) << "Bad algorithm type " << algo << endmsg;
        throw BadSelection("Bad algorithm type");
    }
    ... do other stuff here ...
}

Result MuoHitReco::processEvent(Event& event)
{
    // UnpDataChunk selection remained unchanged from the current code
    UnpChunkSelector selec(D0MCH::MUO_FE);
    const TKey<UnpDataChunk> unpKey(selec);
    THandle<UnpDataChunk> unpchunk=unpKey.find(event);

    if (unpchunk.isValid()==false)
    {
        error_log(ELwarning,"missing data") << "No UnpDataChunk found in event" << endmsg;
        return Result::success; // because no error policy defined
    }

    vector<PDTChannel*> pdt_chans;
    vector<MDTChannel*> mdt_chans;
    extractChannels(unpdata,pdt_chans); // fills pdt_chans
    extractChannels(unpdata,mdt_chans); // fills mdt_chans

    auto_ptr<MuoHitChunk> chunk( new MuoHitChunk );

    try {
        _hit_builder->buildWireHits(pdt_chans, mdt_chans, chunk->wireHitContents());
    }
    catch(const exception& e) {
        // log message or do something interesting
    }

    insertChunk(event,chunk);
}
```

5 Coding Recommendations

5.1 Memory Management

Our recommendation is to change from methods and functions returning newly created objects by pointer to using a fill method. It is easiest to illustrate this through an example:

```
// old create-on-heap method
MuoHitCollection* processData(Thandle<UnpDataChunk> unp) {
    ... do stuff ...
    return new MuoHitCollection(...);
}

// old method use
void func(...) {
    ... do stuff ...
    Thandle<UnpDataChunk> unpdata = ... find chunk in event ...
    MuoHitCollection* hits = processData(unpdata);
    ... use it, copy info to chunk and delete it ...
    MuoHitChunk chunk(*hits);
    ... insert chunk into event
    delete hits;
}

// new fill method
void processData(Thandle<UnpDataChunk> unp, WireHitCollection& hits) {
    hits.clear();
    for each hit found {
        ... do stuff ...
        hits.push_back( WireHit( ... ) );
    }
}

// new fill method use
void func(...) {
    Thandle<UnpDataChunk> unpdata = ... find chunk in event ...
    MuoHitChunk chunk;
    ProcessData(unpdata, chunk.wireHits() );
}
```

The fill method forces the lower-level functions and methods to put the things they generate directly into the container where they belong, instead of into an intermediate place. This style of memory management removes the need for D0OM versions of classes and also removes a large amount of the copying that needs to go on in the current code. It also reduces the number of interactions with the heap manager and makes it clear who the owner of the memory is.

5.2 Use of UnpDataChunk

It has been recommended that Mike Fortner add a couple of utilities to make working with the UnpDataChunk simpler for the user. What drove this recommendation was the code in the MuoHitProcessor that copies PDTChannels and MDTChannels out of the UnpDataChunk. Here we recommend using the new utilities, as well as not copying the channels and using the vector of channel pointers that is returned from the new utilities. Here is a quick outline of what has been proposed and an example use of it.

```
  void example_func(THandle<UnpDataChunk> unpdata)
  {
      …
      vector<PDTChannel*> pdt_chans;
      vector<MDTChannel*> mdt_chans;
      extractChannels(unpdata,pdt_chans); // fills pdt_chans
      extractChannels(unpdata,mdt_chans); // fills mdt_chans

      // or one can do…
      vector<PDTModule*> pdt_mods;
      vector<MDTModule*> mdt_mods;
      extractModules(unpdata,pdt_mods);
      extractModules(unpdata,mdt_mods);
      …
  }
```

The exact name of the utilities and arguments to them will be up to Mike Fortner to decide. This simple utility reduces the code in MuoHitProcessor.cpp to just a few lines that are easy to understand and clean.

5.3 General

5.3.1 Const private members

We have noticed that many classes contain private const data members (of things other than pointers and references). For references or pointers this makes sense, especially if the class does not own the instances and just looks at them. For built-in types such as int and double, and for instances that are owned by the class, this does not make any sense. After all, only the class itself can modify the variables in the private section anywhere, so why put this extra restriction in place. An example is MuoSegment. Here none of these variables need to be const.

```
class MuoSegment {
  …
  private:
    const MuoIndexVector  _muoIndexVector;
    const Position        _position;
    const Direction       _direction;
    const double          _quality;
    double                _angleDr;
  };
```

Furthermore, methods or functions that take const build-it types as arguments should be changed. Doing this has no effect and the compiler is supposed to warn you about it. Unfortunately these warning have been turned off at D0. An example is:

```
  extern bool TDTAToDrDistAxDistPDT(const double td, const double ta,
                    const double angle,
                    double& drift_dist, double& axial_dist);
```

The const double here serves absolutely no purpose and should be changed to just double.

5.3.2 Using THandle<>

When using object retrieved from the event, do not remove the object from the THandle<> class that surrounds it. Use the smart pointer THandle<> as it should be used.

5.3.3 Argument Names

Use descriptive names for arguments to methods and functions, especially if all the arguments are doubles. Here is an example from MuoHitProcess.cpp where we could not tell if there was an error or not:

```
…
// code from MuoHitProcessor
bool time_conversion = T1T2ToTDTAPDT(t1,t2,wire_length,td,ta);
if(time_conversion && TDTAToDrDistAxDistPDT(td, fabs(ta),angle,xd,xa))
…

// function prototypes from muo_util package:
extern bool T1T2ToTDTAPDT(const double t1,
              const double t2,
              const double wirelength,
              double& ta,
              double& td);

extern bool TDTAToDrDistAxDistPDT(const double td,
                  const double ta,
                  const double angle,
                  double& drift_dist,
                  double& axial_dist);
```

Look closely at the arguments ta and td of T1T2ToTDTAPDT(), in the prototype and the code that uses it. They appear to be switched. This is difficult to detect using the compiler because both are doubles. More descriptive names could help. Event better would be to create a simple object that takes the two timing values as arguments and produces the values ta and td using methods:

```
class PDTt1t2TotdtaConverter // this is not a good name
{
public:
 PDTt1t2TotdtaConverter(double t1, double t2):_t1(t1),_t2(t2) { }
 double getTAxial(double wirelength);
 double getTDistance(double wirelength);
private:
 double _t1, _t2;
};
```

Now it is difficult to make a mistake and get the wrong values. The example changes to PDT hit processor use this method.

5.3.4 Illegal Constructs

Goto statements should never be used.

5.4 Error Reporting

Exceptions should be used for reporting and generating errors. In the problem section above, we refer to both global and local error reporting. The framework does not have a policy defined for handling standard exceptions thrown by packages (global errors); it currently just prints the message in the exception and exits. The muon packages need to define exceptions that will be used for local error reporting and use them internally. The error logger can be used directly by the exception classes or used by the algorithm code to issue warnings or important informational messages. It is important to note that the error logger cannot alter program flow, its purpose is to record errors by level and type. All use of error codes should be removed.

5.4.1 Global Error Handling

Here we would like to present some preliminary ideas of how global errors can be handled. The most obvious global error case is the detection of bad data in the event. The action here could be to throw out the event completely or put it into a bad event file. In any case, further processing should not be done on this event. A standard set of base exception classes can be made available by the framework, such as "ScrapEvent", "KillProgram", and "ByPassThisDetector". Algorithms would create their own exception by deriving from these base classes. The framework should not dictate the actions of these exceptions, instead, RCP values should define what happens when the framework catches these exceptions. The current policy of always returning "Success" to the framework from a package is not adequate.

5.4.2 Local Error Handling

New exceptions introduced by the muon packages should derive from the STL exception classes. All of these exceptions are required to produce a text string that explains the error that occurred. Use of exceptions is a nice way to propagate errors up a call stack without constantly using if-then-else structures and return codes at all levels. The example MuoHitProcessor code that we provided shows a simple use of exceptions. Here is an example of using the error logger to report a warning:

```
error_log(ELwarning,"bad_data”)
       << "No time info in channel " << *iter << endmsg;
```

The framework sets up the variable error_log as part of the protected section of the Package class. You may need to create a private error_log variable in low-level algorithm code. To create a private error_log variable, one can do the following:

```
#include "ErrorLogger/ErrorLog.h"
…
ErrorLog error_log(“MuoHitProcessor”); // package/algorithm name
```

Below is an example of a simple exception.

```
#include <stdexcept>
class MuoBadData : public runtime_error {
public:
 MuoBadData(string package, string text):
      runtime_error(type() + package + text) { }
 MuoBadData(ErrorLog& e, string package, string text):
      runtime_error(type() + package + text)
      { e(ELerror,type()) << text << endmsg; }
Private:
 static string type() { return “MuoBadData”; }
};
```

This example optionally logs a message to the error logger (in a very simplistic manner). The exceptions should be throw at the point where the error occurs. The exception should be caught at a level appropriate for handling the error. In the MuoHitProcessor code, it could very well be at the reconstructor level. This means that intermediate levels do nothing about the error – it will be propagated through this layer of code. Doing this makes the code more manageable and easier to follow.

5.5 Package Organization

All *using* declarations (example: *using edm::ChunkID*) must be removed from headers. (They are fine in *.cpp* files). Putting using declarations in the header files opens up the namespaces to everyone that includes the header file. In all cases this is not desirable.

Extraneous headers should be removed (example: *MuoTrackChunk.hpp* includes 11 headers; 5 are not needed). In many cases, long build times can be attributed in part to this. Including header files that are not really needed causes unnecessary dependency generation and hence, unnecessary rebuilding of files. Coupled to this is the use of forward declarations. Forward declarations of classes should be used in header files wherever possible. The only time a header file for another class needs to be included is if an instance of the class is present in the class that is being defined. If other classes are used by pointer or reference only, they can be forward declared.

6 Physical Design

At a high-level the physical design appears to be adequate. An important objective here is to use the exact same algorithm code in offline and in level3. Unfortunately we see that in several cases the algorithm functions are contained in the same implementation files as the code that plucks the information from the event. Doing this really defeats the purpose of breaking the problem apart. We have heard that level3 may want to do releases separate from the offline, if this is the case, then common or shared algorithm pieces really need to live in separate packages or libraries.

The segment reconstructor lives in the same package as the segment finding algorithm. This organization will not allow the algorithm to be used in level3, where reconstructors do not exist. The track reconstructor package likely has the same problems.

7 Documentation

The documentation in these packages is better then we found elsewhere. Some of the UML diagram have incorrect use of symbols and relationship; this was a small point of confusion. Some of the algorithm code is commented quite heavily and this was essential to figuring out what the algorithm actually did. Unfortunately a few of the comments were inconsistent with code and in several cases explained what was going on in terms of variables such as xa, t1, and t2. It would have been much more useful to have one block of comments at the top of the file, using the MuoHitProcessor as an example, explaining the algorithm in terms of the physical hardware that is manipulates and in terms of a mathematical formula. We needed to find an expert in muon geometry and hardware readout to understand this code.

8 Testing

Providing component tests can really prevent d0reco from dying on silly errors that could have been detected and corrected very early on. There is likely to be much finger pointing when d0reco crashes, component tests can be used to show that you are not to blame. If you are to blame for the crash, then the test programs can aid in the discovery of the actual problems. Component tests do not need to be complex and exercise every little bit of code. Do whatever you can to get tests in this code – use the CTBuild model or use the test directory model for a set of executables that test parts of the system.

It was mentioned in the review meeting that in some cases test data is needed and that this is typically difficult to get and use. We will again use the MuoHitProcessor as an example. One way to test this would be to collect a bunch of channel data and run the algorithm on it, comparing the results with what you discovered by hand. Another way would be to prepare a channel with information that you already know the outcome. This is quite simple to do. Now prepare a test that attempts to find the hit or hits in this single fabricated channel. If a test like this cannot easily be done, then the design of the code is wrong. You should be able to exercise the core parts of the algorithm separately if the problem has been decomposed properly.

9 Conclusion

We did not have the time to walk through the segment and track finding code to the same degree as the hit finding code. We believe it suffers from the same problems. One small clue is the code appearance on paper. A subroutine or method that has many nested levels of braces for if-then-else and do-while structures is likely to suffer from decomposition problems. It is likely that the subroutines and methods in question are doing too much – stepping out of their realm of expertise. Any further analysis or review of this code should be done interactively with the authors. We would really like to have gone through the segment/track finding code to make recommendation about marking hits and using STL for selections and sorting.

The examples prepared for this document were not walked through in any formal fashion and are not guaranteed to be completely accurate in respect to the hit finding algorithm (or any other algorithm for that matter). Please notify the authors if errors or inconsistencies are found. We wanted to release this document as soon as possible.